Names, Bindings, and Scope

Copyright © 2018 Pearson. All rights reserved.

Chapter 5 Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Scope
- Scope and Lifetime
- Referencing Environments
- Named Constants

Copyright © 2018 Pearson. All rights reserved.

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables are abstractions of memory cells in Memory.

- Memory is byte-addressable. A variable-sized integer may use several cells.

Copyright © 2018 Pearson. All rights reserved.

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor

- Fundamental elements in terms of variable design include:
  - type, scope, lifetime, type checking, and initialization

Copyright © 2018 Pearson. All rights reserved.

Names or identifiers

- Design issues for names:
  - Are names case sensitive?
  - Are special words reserved words or keywords?

Copyright © 2018 Pearson. All rights reserved.

Names (continued)

- Length
  - If too short, they cannot be connotative
  - Language examples:
    - C99: no limit, but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C89: limit, you need to declare variables at the beginning, not things like for (int i=0; i<5; i++)
    - C# and Java: no limit, and all are significant
    - C++: no limit, but implementers often impose one

Copyright © 2018 Pearson. All rights reserved.

Names (continued)

- Special characters
  - PHP: all variable names must begin with dollar signs
  - Perl: all variable names begin with special characters, which specify the variable's type

```
$name = "Luis"; @colors = ("red", "green", "blue");
%ages = ("Alice" => 25,  "Bob"   => 30);
```

  - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Copyright © 2018 Pearson. All rights reserved.

Names (continued)

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - Names in the C-based languages are case sensitive
    - Names in others are not: SQL, basic, COBOL, Pascal
    - Worse in C++, Java, and C# because predefined names are mixed case (e.g. `IndexOutOfBoundsException`)

Copyright © 2018 Pearson. All rights reserved.

Names (continued)

- Special words

  –A *reserved word* is a special word that cannot be used as a user-defined name

  ```
  int if = 5;
  ```

  –Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has 300 reserved words!)

Copyright © 2018 Pearson. All rights reserved.

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Copyright © 2018 Pearson. All rights reserved.

Variables Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated
  - A variable may have different addresses at different times during execution
  - A variable may have different addresses at different places in a program
  - If two variable names can be used to access the same memory location, they are called aliases
  - Aliases are created via pointers, reference variables, C and C++ unions
  - Aliases are harmful to readability (program readers must remember all of them)

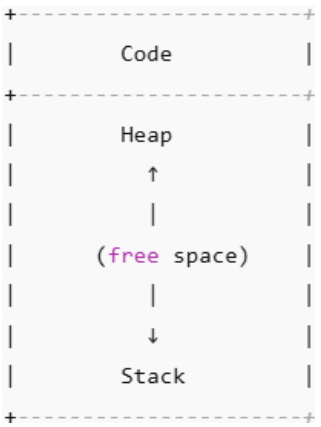
Copyright © 2018 Pearson. All rights reserved.

Variables Attributes (Examples)

Where is the variable x located?

User address space? r real address in RAM ?

```
void foo() {
   int x = 10;   // local variable
  }
```

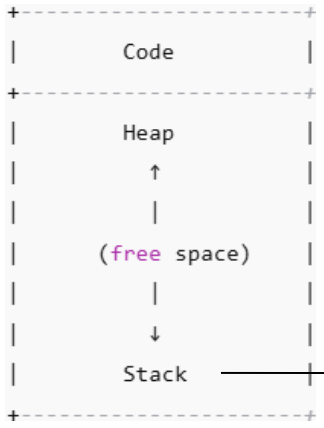


Copyright © 2018 Pearson. All rights reserved.

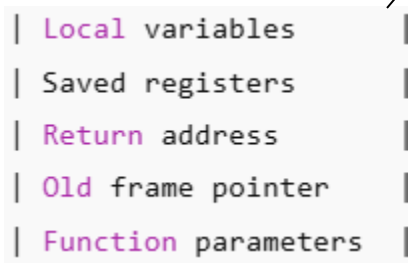
Variables Attributes (Examples)

Example

```
int main() {
    foo();
    foo();
    foo();
}
```



Stack frame



Running Stack
Concept of an evolving stack

Call 1 to foo():
Stack top → [x | return addr | old FP]

Return from foo():
Stack frame removed

Call 2 to foo():
Stack top → [x | return addr | old FP] ← may be at a different address (FP frame pointer)

Copyright © 2018 Pearson. All rights reserved.

Variables Attributes (Aliases)

```
int main() {
    int x = 10;
    int *p = &x;   // p points to x
    int *q = &x;   // q ALSO points to x (alias)
    *p = 20;
    *q = 5; overwrites the same memory location.
}      // change x through p
```

Copyright © 2018 Pearson. All rights reserved.

Variables Attributes (continued)

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision

- *Value* - the contents of the location with which the variable is associated

  - The l-value of a variable is its address
  - The r-value of a variable is its value

- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

Copyright © 2018 Pearson. All rights reserved.

The Concept of Binding

A *binding* is an association between an entity and an attribute, for instance:

Variable -> Type

Variable -> Value

Operation -> Symbol

- *Binding time* is the time at which a binding takes place.

Copyright © 2018 Pearson. All rights reserved.

Possible Binding Times

- Language design time -- bind operator symbols to operations
  Example: * is bound to the multiplication operation at LDT

- Language implementation time-- bind floating point type to a representation

- Compile time -- bind a variable to a type in C or Java

```
gcc main.c -c
```

What happens here:

- Source code → object file (`.o`)
- Types are checked
- Syntax is checked
- Variables are assigned storage classes (static, automatic, etc.)
- Symbolic addresses/offsets are generated

But the program is **not running yet**.

- Load time -- bind a C or C++ `static` variable to a memory cell)

```
./program
```

Now the OS loader:

- Copies the program into memory
- Assigns real memory addresses to:
  - Code
  - Globals
  - Static variables
- Sets up stack and heap
- Applies relocations

Now variables have **real addresses in RAM**.

- Runtime -- bind a non-static local variable to a memory cell

Copyright © 2018 Pearson. All rights reserved.

Static and Dynamic Binding

- A binding is *static* if it first occurs before run time and remains unchanged throughout program execution. (at compile time)

- A binding is *dynamic* if it first occurs during execution or can change during execution of the program

Copyright © 2018 Pearson. All rights reserved.

Type Binding

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an implicit declaration

Copyright © 2018 Pearson. All rights reserved.

Explicit/Implicit Declaration (TYPE)

- An *explicit declaration* is a program statement used for declaring the types of variables
- int x = 10;
- float y = 3.14;

- An *implicit declaration* is a default mechanism for specifying the types of variables through default conventions, rather than declaration statements
- I = 10 X = 3.14 in FORTRAN, vars I-N are int, otherwise reals

- Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Copyright © 2018 Pearson. All rights reserved.

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
  - C# - a variable can be declared with **var** and an initial value. The initial value sets the type

```
var price = 19.99;      // double
```

  - Visual Basic 9.0+, ML, Haskell, and F# use type inferencing. The context of the appearance of a variable determines its type

```
Dim price = 19.95     ' Inferred as Double
```

Copyright © 2018 Pearson. All rights reserved.

Dynamic Type Binding

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited)) Type checking happens at runtime
- Specified through an assignment statement

  e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Copyright © 2018 Pearson. All rights reserved.

Variable Attributes (continued)

- Storage Bindings & Lifetime
  - Allocation - getting a cell from some pool of available cells
  - Deallocation - putting a cell back into the pool

- The lifetime of a variable is the time during which it is bound to a particular memory cell

Copyright © 2018 Pearson. All rights reserved.

Categories of Variables by Lifetimes

- Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables in functions,
- Memory is reserved **before the program starts running**
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

Copyright © 2018 Pearson. All rights reserved.

Categories of Variables by Lifetimes

```
#include <stdio.h>
int counter = 0;   // static storage binding

void foo() {
   counter++;
   printf("counter = %d, address = %p\n", counter, (void*)&counter);
}
int main() {
   foo();
}
#include <stdio.h>
```

```
void foo() {
   static int x = 0;   // static storage binding
   x++;
   printf("x = %d, address = %p\n", x, (void*)&x);
}

int main() {
   foo();
}
```

Copyright © 2018 Pearson. All rights reserved.

Categories of Variables by Lifetimes

- Stack-dynamic--Storage bindings are created for variables when their declaration statements are *elaborated*.

```
int main() {
   int x = 10;
}
```

•Statically bound in type (The type of x is known and fixed at compile time Dynamically bound in storage)

Elaboration = **storage allocation + binding**

- If scalar, all attributes except address and value are statically bound
  - local variables in C subprograms (not declared **static**) and Java methods
- Advantage: allows recursion; conserves storag
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

```
void counter() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}
```

```
void counter() {
    int count = 0;   // stack dynamic
    count++;
    printf("%d\n", count);
}
```

Copyright © 2018 Pearson. All rights reserved.

Categories of Variables by Lifetimes

- *Explicit heap-dynamic* – int *p = malloc(sizeof(int)); *p = 10;
- Nameless, just a **raw memory cell in the heap, *p is in the stack**
- **int *p; Pointer variable**
- **malloc(...) Heap memory cell**

- Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution

- Referenced only through pointers or references, e.g. dynamic objects in C++ (via **new** and **delete**), all objects in Java
- **Integer obj = new Integer(5); in java**

- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

Copyright © 2018 Pearson. All rights reserved.

Categories of Variables by Lifetimes

- *Implicit heap-dynamic*—variables whose memory is allocated on the heap automatically, without the programmer explicitly requesting it, and whose type may be determined at runtime.

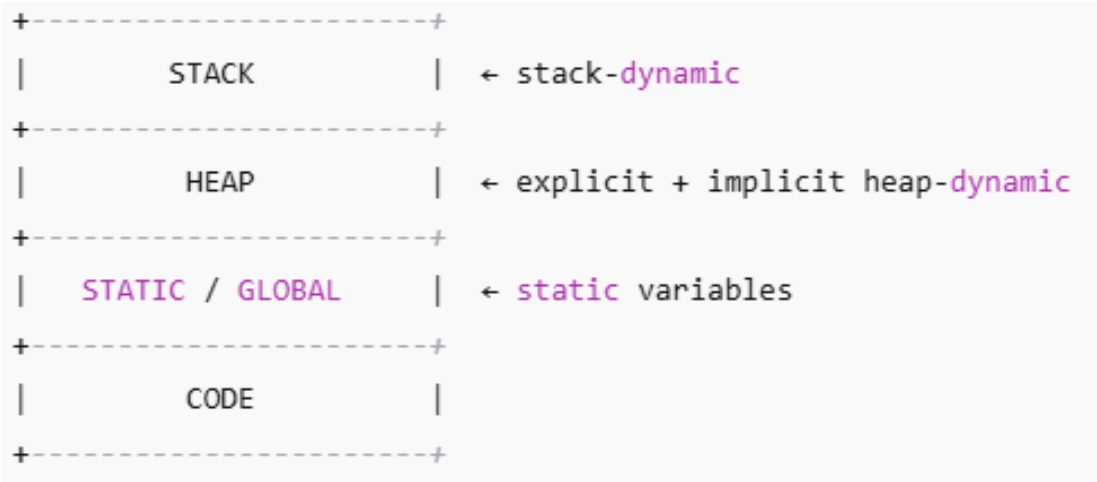
- Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP. Example in Python

```
x = 10; x= [“a”,”b”]
y = "hello"
z = [1, 2, 3]
```

- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Copyright © 2018 Pearson. All rights reserved.

Visual process/variables



Copyright © 2018 Pearson. All rights reserved.

Feature	Static	Stack-Dynamic	Explicit Heap-Dynamic	Implicit Heap-Dynamic
Where stored	Data segment	Runtime stack	Heap	Heap
When storage is bound	Before execution (load time)	At runtime when declaration is elaborated (function/block entered)	At runtime when programmer requests it	At runtime automatically when assigned
Who controls allocation	Compiler / loader	Runtime system (function call)	Programmer (`new`, `malloc`)	Language runtime / interpreter
Who controls deallocation	Automatic at program end	Automatic at function/block exit	Programmer (`free`, `delete`) or GC	Garbage collector / runtime
Lifetime	Entire program	Duration of function/block	Until explicitly freed (or GC)	Until no references remain
Can share storage between calls?	❌ No	✅ Yes (stack reused)	Sometimes (if freed and reused)	Managed automatically
Supports recursion?	❌ No (for locals)	✅ Yes	✅ Yes	✅ Yes
Memory efficiency	Low (always allocated)	High	Medium (depends on programmer)	Low–Medium (GC overhead)
Accessed how	Directly by name	Directly by name	Through pointers/references	Through references
Type binding	Static	Static	Static	Often dynamic
Examples of languages	FORTRAN (early), C globals	C, C++, Java locals	C (`malloc`), C++ (`new`)	Python, JavaScript, Ruby
Example code	`static int x;`	`int x;` inside function	`int *p = malloc(sizeof(int));`	`x = [1,2,3]` (Python)

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables

```
x = 10

def f():
    x = 5
    print(x)

f()
```

Copyright © 2018 Pearson. All rights reserved.

Static (lexical) Scope

- Based on program text
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*
- Some languages allow nested subprogram definitions, which create nested static scopes (e.g., Ada, JavaScript, Common Lisp, Scheme, Fortran 2003+, F#, and Python)

Copyright © 2018 Pearson. All rights reserved.

Static (lexical) Scope

```
function big() {

   function sub1() {
     var x = 7;
     sub2();
   }

   function sub2() {
      var y = x;
   }

  var x = 3;
  sub1();
}
```

What will be the value of y ?

Copyright © 2018 Pearson. All rights reserved.

Perl offers dynamic scope option

```
------------------Perl --------
our $x = 10;
  sub foo {
      local $x = 20;   # dynamically scoped
      bar();
     }
   sub bar {
    print $x;  # prints 20
  }
_____________________  shell script----

x=10
foo() {
  local x=20
  bar
}

bar() {
  echo $x   # prints 20
}
```

Copyright © 2018 Pearson. All rights reserved.

Scope (continued)

- Variables can be hidden from a unit by having a "closer" variable with the same name (or shadows)

```
x = 10

def f():
    x = 5   # shadows (hides) the global x
    print(x)

f()       # 5
print(x)  # 10
```

Copyright © 2018 Pearson. All rights reserved.

Blocks

A **block** is a section of code that can contain **its own variable declarations** and therefore creates a **new static (lexical) scope** inside a larger scope.

Blocks in C and C-like Languages: statements inside { })

```
if (list[i] < list[j]) {
    int temp;
    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}
```

In Python, **blocks like if, for, or while do NOT create a new scope**.

```
if True:
    x = 10
print(x)   # still visible
```

Python does not use blocks to create new static scopes. Instead, functions define scopes each function introduces a new local scope, and nested functions create nested lexical scopes. Same with JavaScript

Copyright © 2018 Pearson. All rights reserved.

The **LET** Construct

- Most functional languages include some form of **`let`** construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (
  (name1 expression1)
  …
  (namen expressionn)
)
```

Copyright © 2018 Pearson. All rights reserved.

The **LET** Construct (continued)

- In ML:

```
let
  val name1 = expression1
  …
  val namen = expressionn
in
 expression
end;
```

- In F#:
  - First part: **let** left_side = expression
  - (left_side is either a name or a tuple pattern)
  - All that follows is the second part

Copyright © 2018 Pearson. All rights reserved.

Declaration Order

- In older C (C89), all variable declarations in a function (except inside nested blocks) had to appear **at the beginning** of the function

```
void foo() {
   int x;
   int y;   // all declarations first
 x = 5;   // statements only after
}
```

```
for (int i = 0; i < 5; i++) // not allow in C89
```

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
-

```
for (int i = 0; i < 5; i++) // legal in C99
```

  - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
  - In the official documentation of C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
    - However, that is misleading, because a variable still must be declared before it can be used

Copyright © 2018 Pearson. All rights reserved.

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
  - The scope of such variables is restricted to the `for` construct

Copyright © 2018 Pearson. All rights reserved.

Global scope

- Some languages (C, C++, PHP, JavaScript, Python) allow variables to be defined **outside any function**. These are **global variables** and may be visible inside functions depending on the language's rules.

- C and C++ distinguish between **declarations** and **definitions** of global variables: **Declarations** describe type and attributes but do not allocate storage. **Definitions** allocate storage

Copyright © 2018 Pearson. All rights reserved.

Global scope

- Some languages (C, C++, PHP, JavaScript, Python) allow variables to be defined **outside any function**. These are **global variables** and may be visible inside functions depending on the language's rules.

- C and C++ distinguish between **declarations** and **definitions** of global variables: **Declarations** describe type and attributes but do not allocate storage. **Definitions** allocate storage

- **Declaration** extern int count; // declaration only
- **Definition** count = 0; // create a variable and allocates memory

Copyright © 2018 Pearson. All rights reserved.

Global scope (example)

hello.c
```
Int x =10;
```

hello.h
```
extern int x;
```

luis.c
```
#include “hello.h”
x=x+5;
printf(“x=%d\n”,x); /prints 15, update global
```

---

hello.c
```
#include <stdio.h>

void greet() {     // Definition
   printf("Hello!\n");
}
```

hello.h
```
void greet() or extern void greet()
```

luis.c
```
#include “hello.c”

greet();
```

Extern in functions

hello.c
```
add(int a, int b) {
      return a + b;
}

}
```

hello.h
```
extern add(int a, int b)
```

luis.c
```
#include “hello.h

int main() {
int x = add(3,4);
```

Copyright © 2018 Pearson. All rights reserved.

Global scope python

- Python allows globals to be **referenced** inside a function, but to **assign** to a global variable inside a function, it must be declared global in that function. Otherwise, the assignment creates a **new local variable**, which makes earlier references illegal

```
day = "Monday"

def tester():
    global day
    print(day)      # OK
    day = "Tuesday" # modifies global

tester()
print(day)  # Tuesday
```

```
Monday
Tuesday
```

Copyright © 2018 Pearson. All rights reserved.

Global Scope (continued)

- PHP
  - Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions
  - The scope of a variable (implicitly) declared in a function is local to the function
  - The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
    - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Copyright © 2018 Pearson. All rights reserved.

```php
<?php
$greeting = "Hello";   // Global variable (outside any function)
?>

<p>Before function call</p>

<?php
function sayHello() {
    $name = "Luis";   // Local variable (only inside function)

    // Access global using GLOBAL keyword
    global $greeting;
    echo "$greeting $name (using global)<br>";

    // Access global using $GLOBALS array
    echo $GLOBALS['greeting'] . " again $name (using GLOBALS)<br>";
}
?>

<p>Calling function...</p>

<?php
sayHello();
?>

<p>After function call</p>

<?php
echo "Outside function, greeting is still: $greeting<br>";
?>

<p>Calling function...</p>

<?php
sayHello();
?>

<p>After function call</p>

<?php
echo "Outside function, greeting is still: $greeting<br>";
?>

</body>
</html>
```

```
PHP Scope Example

Before function call

Calling function...
Hello Luis (using global)
Hello again Luis (using GLOBALS)

After function call
Outside function, greeting is still: Hello
```

Global Scope (continued)

- Python
  - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

- count = 10 # Global variable. Which function will work

```
def show():
    # Referencing global variable (reading only)
    print("Count is:", count)
```

```
def increment_one():
    # Trying to assign without declaring global
    count = count + 1
```

```
def increment_two():
    global count
   count = count + 1
```

Copyright © 2018 Pearson. All rights reserved.

Evaluation of Static Scoping

- Works well in many situations
- Problems: (make everything global to avoid over thinking)
  - In most cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Copyright © 2018 Pearson. All rights reserved.

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point

Copyright © 2018 Pearson. All rights reserved.

Scope Example

```
function big() {

    function sub1() {
        var x = 7;
    }

    function sub2() {
        var y = x;
        console.log(y);
    }

    var x = 3;

    sub1();
    sub2();
}

big();
```

Static scoping
» Y=
Dynamic scoping
» Y=

```
function big() {

    function sub1() {
        var x = 7;
        sub2();           // sub1 calls sub2
    }

    function sub2() {
        var y = x;
        print(y);
    }

    var x = 3;

    sub1();
}

big();
```

Static scoping
Y=
Dynamic
scoping
Y=

Copyright © 2018 Pearson. All rights reserved.

Scope Example

```
function big() {

    function sub1() {
        var x = 7;
    }

    function sub2() {
        var y = x;
        console.log(y);
    }

    var x = 3;

    sub1();
    sub2();
}

big();
```

Static scoping
» Y=3
Dynamic scoping
» Y=3

```
function big() {

    function sub1() {
        var x = 7;
        sub2();           // sub1 calls sub2
    }

    function sub2() {
        var y = x;
        print(y);
    }

    var x = 3;

    sub1();
}

big();
```

Static scoping
Y=3
Dynamic scoping
Y=7

Copyright © 2018 Pearson. All rights reserved.

Scope Example

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check (With **static scope**, the compiler can look at the program text and decide)
    3. Poor readability- it is not possible to statically determine the type of a variable

Copyright © 2018 Pearson. All rights reserved.

Evaluation of static scope evaluation

Impossible to statically type check (With **static scope**, the compiler can look at the program text and decide). Example of static scope

```
x = 10
def f():
    return x + 1
```

The type checker knows: x refers to the global x

type of x = int

So, x + 1 is valid

All of this can be determined **without running the program**.

Copyright © 2018 Pearson. All rights reserved.

Scope and Lifetime

- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a **`static`** variable in a C or C++ function

Copyright © 2018 Pearson. All rights reserved.

Example (scope and lifetime)

```c
#include <stdio.h>

void counter() {
    static int count = 0;  // static local variable
    count++;
    printf("count = %d\n", count);
}

int main() {
    counter();
    counter();
    counter();
    return 0;
}
```

```
count = 1
count = 2
count = 3
```

```c
void counter() {
    static int count = 0;
}

int main() {
    printf("%d", count);
}
```

?

Lifetime?

What is static scope?

What happens if we remove the static
From the first function counter()?

Copyright © 2018 Pearson. All rights reserved.

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- A subprogram is active if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms

Copyright © 2018 Pearson. All rights reserved.

Examples: *referencing environment*

```python
x = 1

def outer():
    y = 2
    def inner():
        print(x + y)
```

At the line `print(x + y)`, the referencing environment is:

Scope Level	Names Visible
Local to `inner`	(none declared here)
Enclosing (`outer`)	`y`
Global	`x`
Built-in	`print`, etc.

```
function A() {
    x = 5
    B()
}

function B() {
    y = 10
    C()
}

function C() {
    print(x + y)
}
```

`C → B → A`

So the referencing environment includes:

From	Variables
C (local)	none
B (caller)	y
A (caller's caller)	x

So again `{x, y}` — **but only because of the call chain**, not because of code nesting.

If `C()` were called directly from `main`, `x` or `y` might be missing.

Copyright © 2018 Pearson. All rights reserved.

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage (an identifier that represents a value that **does not change during program execution)**

- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - C++ and Java: expressions of any kind, dynamically bound
  - C# has two kinds, **`readonly`** and **`const`**
    - the values of **`const`** named constants are bound at compile time
    - The values of **`readonly`** named constants are dynamically bound

Copyright © 2018 Pearson. All rights reserved.

Named Constants

- const double PI = 3.14159;
- area = PI * r * r;
- const double INTEREST_RATE = 0.08;
- double interest = balance * INTEREST_RATE;

Static binding

const int SIZE = 10;

int arr[SIZE]; //SIZE must be known at compile time

Copyright © 2018 Pearson. All rights reserved.

Name constants

- const int x = rand(); // Value determined at runtime (dynamic binding)
  Still constant but not known during compilation.

```
const int a = 5;          // compile-time constant
const int b = f();        // runtime constant

final int x = 10;         // compile-time constant if literal
final int y = getValue();  // runtime constant
```

Copyright © 2018 Pearson. All rights reserved.

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Copyright © 2018 Pearson. All rights reserved.